Attention is all you need

Introduction and Discussion

Sequence modeling and transduction are important problems in Natural Language Processing. Sequence Modeling is the task of predicting what word or letter comes next. The current output is dependent on the previous input and the length of the input is not fixed.
Almost all state-of-the-art results about sequence modeling and transduction are achieved with recurrent neural network such as LSTM (Long Short-Term Memory) or gated recurrent neural networks.

What is **recurrent neural network** ? RNNs are a generalization of feedforward neural network that have an internal memory. They learn during training and they also remember what they have learnt from previous inputs while generating output. Indeed, the output is influenced by using input and *hidden states*. Hidden states act like memory, they represent the context based on prior inputs/outputs. At position t : each hidden state h_t is a function of the previous hidden state h_{t-1} and the input for t. So, different output can be produced by the same input depending on previous input in the series.
However RNNs have some disadvantages :
- The gradient could explode (or vanish) if the derivatives are large (or small respectively), this is due to a large number of multiplications of the derivatives.
- Long sequences cannot be processed with a tanh or a relu activation function.
- It is a difficult task to train an RNN.

Long Short-Term Memory : LSTM have cells state, these cells help to transfer relative information, they are a sort of "memory" of the network. Each cell state has gates; their objective is to regulate the flow of information. These gates select the data that is important to keep and throw away the data that is not relevant. To do this, they use the sigmoid function, the function returns a value between 0 and 1, if the returned value is close to 1 the data is kept otherwise if the returned value is close to 0, the data is forgotten.
The problem of short-term memory is solved, thanks to the cells state and their gates, they allow information from the earlier time steps to make their way to later time steps.
LSTM has 3 gates :

1. Forget gate – This gate decides what information should be discarded. We pass the previous hidden state and information from the current input to the sigmoid function, which return a value between 0 and 1.
2. Input gate – First the current input and the previous hidden state are passed through the sigmoid function. Then, they are passed to the tanh function which regulates the network by giving a weight ranging from -1 to 1. Finally, the output of the sigmoid and the tanh function are multiplied so that, the sigmoid function decides which information is important to keep from the tanh output.
3. Output gate – The output gate determines the next hidden state. The sigmoid function takes the previous hidden state (that contains the information on the previous inputs), and the current input. Then, the modified cell state is passed through the tanh function, and we multiply the sigmoid and the tanh output to get the next hidden state.

So, the equations for the forward pass of an LSTM unit with a forget gate are:

$$
\begin{aligned}
f_t &= \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
\widetilde{c_t} &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
c_t &= f_t \cdot c_{t-1} + i_t \cdot \widetilde{c_t} \\
h_t &= o_t \cdot \tanh c_t
\end{aligned}
$$

Gated Recurrent Units (GRU) : GRU are very similar to LSTM. The difference is that GRU use only hidden state as the memory (there is no cell state)**.** Besides, GRU have two gates :

1. Update gate – Its role is similar to the role of the forget and input gate of an LSTM. It decides what information to keep, and what information to discard.
2. Reset Gate – It decides how much of the past information to forget.

To calculate the next hidden state we use the formulas :

$$
\begin{aligned}
z_t &= \sigma(W_z x_t + U_z h_{t-1} + b_z) \\
r_t &= \sigma(W_r x_t + U_r h_{t-1} + b_r) \\
\widehat{h_t} &= \tanh(W_h x_t + U_h(r_t \cdot h_{t-1}) + b_h) \\
h_t &= z_t \cdot \widehat{h_t} + (1 - z_t) \cdot h_{t-1}
\end{aligned}
$$

RNN Encoder – Decoder with attention (Bahdanau et al):
Bahdanau proposed an attention mechanism as a neural machine translation system improvement (today, attention is also used in many other applications as computer vision, speech processing…)
Encoder and decoder are stacks of RNN units. Basic encoders read the input sentence, and encode it into a context vector *c*. Basic decoder produce the words in a sentence one after another using the context vector *c*.
In encoder-decoder with attention mechanism : The encoder uses a bidirectional RNN, in order to get an annotation that summarizes the preceding and the following words of each word of the input sentence. The forward RNN $\vec{f}$ calculates a sequence of forward hidden states $(\overrightarrow{h_1}, \ldots, \overrightarrow{h_{T_x}})$, and the backward RNN $\overleftarrow{f}$ calculates a sequence of backward hidden states $\overleftarrow{h_1}, \ldots, \overleftarrow{h_{T_x}})$ as a means to get information from preceding and following words. We get an annotation for each word x_j : $h_j = \left[\overrightarrow{h_j}, \overleftarrow{h_j}\right]$.
The decoder uses these annotations to calculate a context vector **for each** output word (unlike the basic decoder that uses a context vector for the entire sentence.)
The context vector c_i for the output word y_i is computed as a weighted sum of these annotations h_i : $c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j$.
The weight α_{ij} is computed by $\alpha_{ij} = \frac{e^{e_{ij}}}{\sum_{k=1}^{T_x} e^{e_{ik}}}$, where $e_{ij} = a\left(s_{i-1}, h_j\right)$ (s_i is an RNN hidden state for time i)., α_{ij} is the probability that the output word y_i is aligned to a source word x_j and e_{ij} is a scoring model which quantifies how the inputs around the word at position j and the output word at position i align.

The problem with the previous methods is that we have to process word by word, thus, it precludes parallelization. Besides for long sentences, the model often forgets the content of

distant positions in the sequence. This paper proposes a model architecture that avoids recurrence and rely only on an attention mechanism : Transformers. Transformers were originally created to solve machine translation problem.

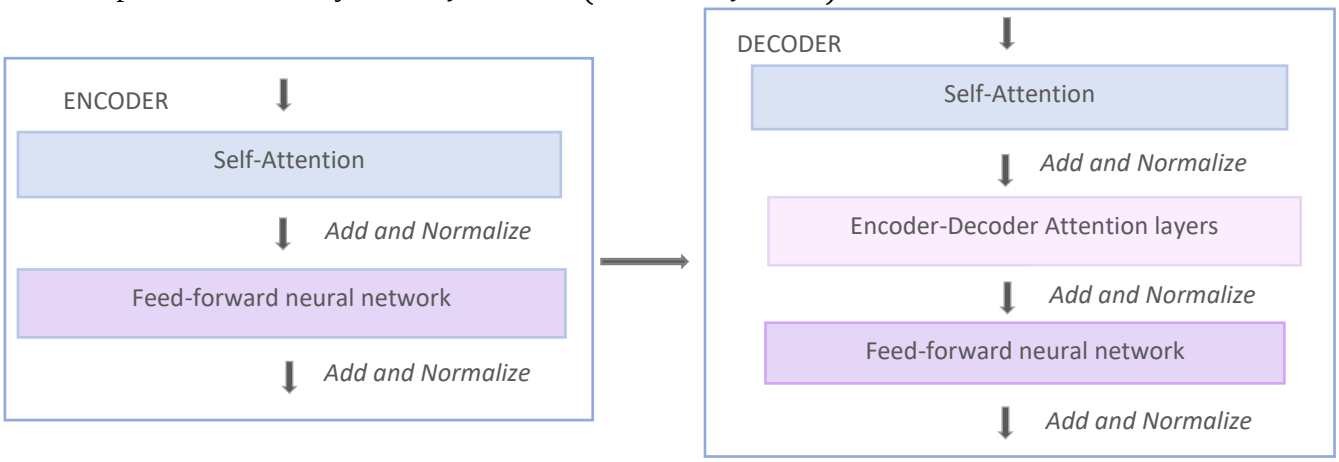
Method

The Transformer uses the architecture of encoder – decoder, with some small changes.
The encoder and the decoder are both composed of a stack of 6 identical layers.

Each encoder consists of two layers: Self-attention and a position-wise fully connected feed-forward neural network. First, the encoder's inputs are passed to the self-attention layer, that looks at other words in the input sentence while encoding a specific word. Then, the output of the self-attention layer is passed to the feed-forward neural network.

Each decoder consists of the previous two layers and a third sub-layer that performs multi-head attention over the output of the encoder stack. The self-attention layer is also modified, it uses a masking function to hide subsequent positions, as the decoder is only allowed to attend to earlier positions in the output sequence.

In addition in the encoder and the decoder each sub-layer has a residual connection around it and is followed by layer normalization.
The output of each sublayer is $LayerNorm\big(x + Sublayer(x)\big)$



In order to pass the input to the encoder, we have to turn each word into a vector using an embedding algorithm. The position of each word and the order of the sequence are important in translation. Thus, we add positional encoding to the embedding of each word.

Self – Attention Layer :

This layer measures the encoding of a word against the encoding of the other words in the sentence and returns a new encoding. There are 6 steps to calculate self-attention.

1. First, we have to create our three matrices : The Query Matrix (Q), the Key Matrix (K) and the Value Matrix (V). To do that, we multiply our three weight matrices with the embedding matrix X (a matrix composed of the embeddings of the words of the input sentence.) The three weight matrices were created during the training process.
   → We get $K = X \cdot W_k$, $Q = X \cdot W_q$ and $V = X \cdot W_V$.

2. Then, we calculate the score that indicates the connection between each pair of word. This score is calculated by multiplying the Query and the Key matrix.
3. The third step is to divide the result by the square root of the dimension of the key matrix.
4. Then, we take the softmax of the result in order to get normalized scores that add up to 1.
5. Finally we multiply the softmax scores with the value matrix.

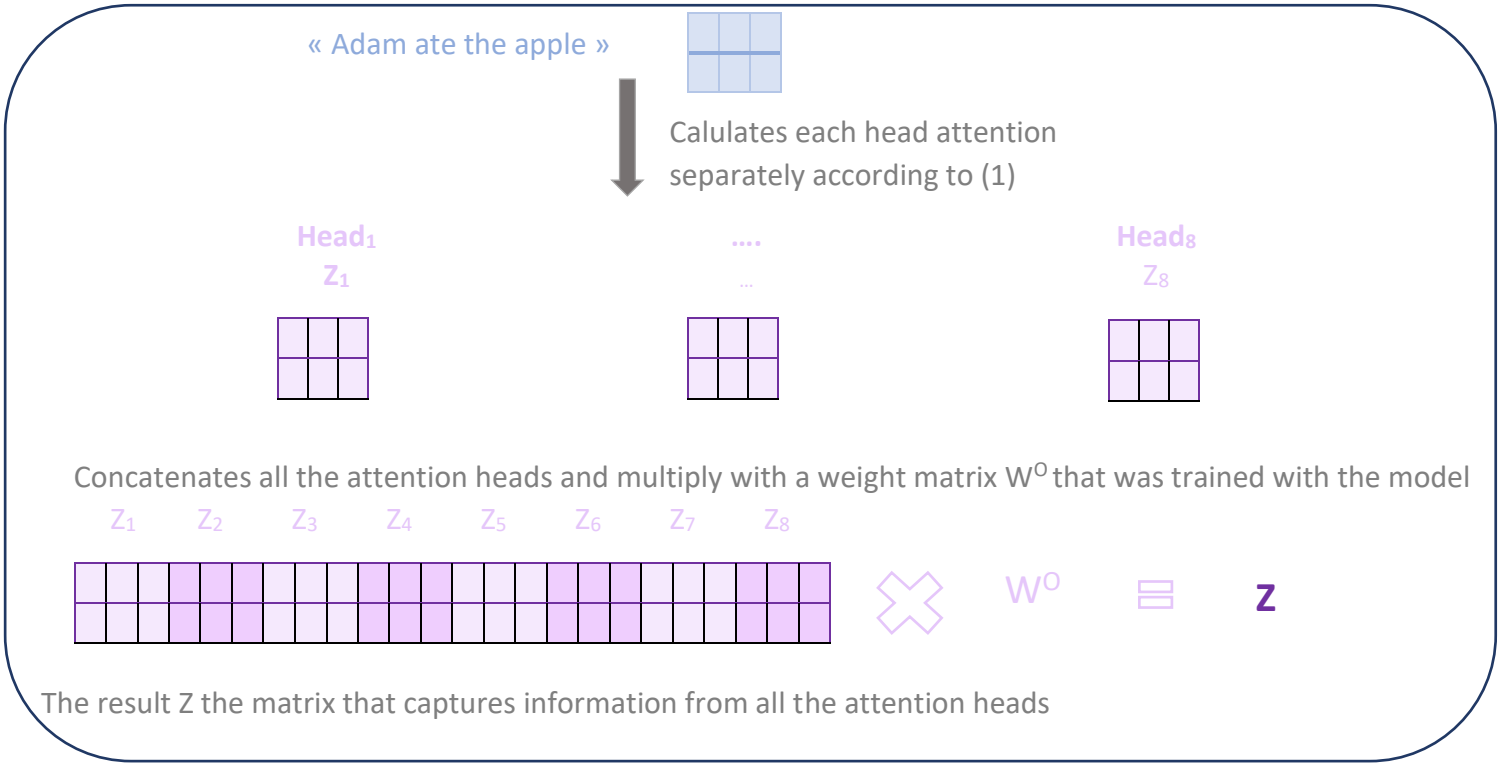
$$Q = X \cdot W_Q$$
$$K = X \cdot W_K$$
$$V = X \cdot W_V$$
$$Z = \frac{Softmax(Q \cdot K^T)}{\sqrt{d_k}} V$$

Multi-Head Attention :

Transformers use Multi-Head Attention, that consists of several attention layers running in parallel. The role of multi-head attention is to allow each word to focus on different positions. Although in a regular self-attention layer each word contains information about all the other words, sometimes it can be dominated by the word itself. If we want to translate the sentence "Adam ate the apple", when we translate the word ate, we want to know who ate the apple to conjugate the verb.
Besides, multi-head attention gives the attention layer multiple representation subspaces. Indeed, we have multiple Query, Key and Value weight matrices that we trained during the training process (in this work they used eight set of weight matrices). Each of this triplet is used to project the input embeddings into a different representation subspace. Thus, after these calculations we get eight different Z matrices. We get the formulas :

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V) \quad (1)$$
$$Multihead(Q,K,V) = Concat(head_1, \dots, head_8)W^O$$

We can illustrate these calculation as follows :



How the encoder and decoder work together ?

The encoder starts by processing the input sentence. Its self-attention layers use multi-head attention to calculate a set of attention vectors Q, K and V. All the keys, queries and values come from the output of the previous layer in the encoder. We can notice that the encoder can also attend to subsequent positions in the previous layer.
Then, the decoder's self-attention layers use multi-head attention in a similar way (Q, K and V come from previous layer in the decoder). The difference is that the decoder is only allowed to attend to previous positions, so all future positions are masked (they are set to be $-\infty$)
Encoder-Decoder Attention layers also use multi-head attention, keys and values come from the output of the encoder, and queries come from previous decoder layers.
This layer allows the transformer to have a better understanding of the context and the complete input sentence using the keys and the values of the encoder.
The Transformer repeats these steps until a symbol indicating the end of the task is reached.

The final linear layer is a fully connected neural network. The linear layer associated with the softmax layer converts the output vector of the decoder into a word.
The linear layer produces a score for each word, and the softmax layer transforms these scores into probabilities, the word with the highest probability is chosen to be the next output.

Evaluation and Results

In the Jupyter notebook, I implemented the basic Transformer for German – English translation.
I trained the transformer with the default parameters : the encoder and decoder stacks layers are composed of 6 layers, d_{model} is set to 512, d_{ff} to 2048, there are 8 attention heads, d_k and d_v are set to 64 and dropout to 0.1.
In the basic implementation we got an accuracy of 82% on the training set and of 62% on the validation set.

1. First, I tried changing the optimizer to AdamW optimizer :
   What is the difference between Adam optimizer to AdamW optimizer ?
   In Adam Optimizer, we derive the sum of the regularization term and the cost function to calculate the gradient, thus if we add the weight decay, the moving averages of the gradient will not only track the gradients but also the regularization term.
   Therefore the authors of the Adam Optimizer proposed a new version where the weight decay is performed after updating the moving averages. Thus, the decay will only be proportional to the weight itself.
   With AdamW optimizer we got a train accuracy of 90 % and a test accuracy of 62.4%.
2. I also tried to train the model with the SGD optimizer, but the results were not relevant, we got a train accuracy of 56.48% and a validate accuracy of 46.56%.
   Jingzhao Zhang et. al. 2020 explained that one of the reasons that Adam outperform SGD is that Adam performs coordinate-wise gradient clipping, so it can tackle heavy-tailed noise unlike SGD.
3. Change the model structure to 32 heads attention, $d_k = 32$ and $d_v = 32$.
   I got an overfitting problem. Indeed the train accuracy is almost perfect 97.44% while the validation accuracy is only 53.73%
4. Then, I changed the structure of the transformer, I use a transformer with ten layers. Although the train accuracy increased to 88.59 %, the validation accuracy decreased to 50.492 %.

5. I also run the model for a French-English translation task, the results were better : accuracy on the training set of 92%, and accuracy on the validation set of 74.25%.
6. Finally I set the dropout to 0.25 for French-English translation. That didn't improve the model, we got accuracy of 80.47% on the training set and of 46.83% on the validation set.

Languages	# Epoch	Variations	Train Acc	Valid Acc	Train PPL	Val PPL
Ge – En	400	-	82.41 %	62.03 %	2.39	13.43
Ge – En	450	AdamW Optimizer	90.74 %	62.41%	1.56	19.15
Ge – En	450	d_k=d_v = heads = 32	97.44%	53.73%	1.098	50.93
Ge – En	450	SGD Optimizer	56.48 %	46.56 %	9.42	22.22
Fr – En	450	-	92.19 %	74.25 %	1.452	7.019
Fr – En	450	Dropout = 0.25	80.47 %	46.83%	2.445	24.109